**1 Spring基本应用——学习笔记**

**目录：** 1 Spring的入门程序 2 Spring中的Bean 3 Spring的AOP技术

1. Spring的入门程序

spring-framework-5.\*.\*.RELEASE-dist.zip，下载URL为：“<http://repo.spring.io/simple/libs-release-local/org/springframework/spring/>”。

Spring开发时需要用的**基本JAR包**：

|  |  |
| --- | --- |
| spring-core.RELEASE.jar | 包含Spring框架基本的核心工具类 |
| spring-beans.RELEASE.jar | 包含访问配置文件、创建和管理Bean以及IoC操作相关类 |
| spring-context.RELEASE.jar | 提供了在基础IoC功能上的扩展服务，如邮件服务、远程访问、缓存 |
| spring-expression.RELEASE.jar | 定义了Spring的表达式语言 |

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Spring的入门程序开发**

**(1)** 在IntelliJ IDEA中，创建一个名为spring01的maven项目， 注意不要选择模板。

**(2)** 修改pom.xml文件，从提供的模板spring-pom.txt中复制粘贴，导入Spring依赖的jar包，首次导入速度比较慢。

**(3)** 在目录src/main/resources下， 创建Spring的配置文件**applicationContext.xml**， 可从提供的模板applicationContext.xml 直接copy。

*Tips：配置文件的模板可以在spring框架的解压文件夹中的doc目录，在spring-framework-reference文件夹下打开html文件夹，找到Core.html文件下的1.2.2小节中的Configuration metadata中找到配置文件的约束信息。初学者只需复制到项目的配置文件中使用即可。*

**(4)**在目录src/main/java下， 创建包jmu.ioc，并在包中创建接口**StudentDAO**。

|  |
| --- |
| package jmu.ioc; public interface StudentDAO { public void say();} |

**(5)**在包jmu.ioc下创建接口实现类**StudentDAOImpl** ，实现接口中的say()方法。

|  |
| --- |
| package jmu.ioc; public class StudentDAOImpl implements StudentDAO{  public void say() {System.out.println("StudentDAO say Hello World!"); } } |

**(6)** 在路径src/main/resources下打开Spring配置文件applicationContext.xml，注册一个id为studentDAO的Bean。

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://www.springframework.org/schema/beans  http://www.springframework.org/schema/beans/spring-beans.xsd">  <bean id="**studentDao**" class="**jmu.ioc.StudentDAOImpl**" />  </beans> |

**(7**) 编写测试程序，打开StudentDAOImpl.java文件， 选择“StudentDAOImpl”文字，按快捷键“**Alt+Enter**”，在弹出的选项中选择“**Create Test**”。 会自动在路径src/**test**/java/jmu/ioc下创建测试文件 **StudentDAOImplTest**.java。

|  |
| --- |
| package jmu.ioc; import org.junit.Test; import org.springframework.context.ApplicationContext; import org.springframework.context.support.ClassPathXmlApplicationContext; import static org.junit.Assert.\*; public class **StudentDAOImplTest** {  **@Test**  public void test1(){  //1.初始化Sprinp容器，加载配置文件  ApplicationContext app =  new ClassPathXmlApplicationContext("applicationContext.xml");  //2.通过容器，获取对象  StudentDAO studentDAO = (StudentDAO) app.getBean("studentDAO");  studentDAO.say();//3.对象调用say()方法  } } |

**依赖注入**的概念 依赖注入(Dependency Injection DI)与反转控制(IoC Inversion of Control)含义相同，不同角度描述而已。

Spring框架的**作用**：  
☆对象的实例不再由调用者创建，而是由Spring容器创建。控制权由应用代码转移到Spring容器，控制权发生了反转。  
☆Spring容器负责将被依赖对象赋值给调用者的成员变量，相当于为调用者注入了它依赖的实例。

依赖注入的**实现方式**

☆属性setter方法注入：通过调用无参数构造器或无参静态工厂方法实例化Bean后，调用该Bean的setter方法，即可实现setter方法的依赖注入。

☆构造方法注入：指IoC容器使用构造方法注入被依赖的实例。基于构造方法的依赖注入通过调用带参数的构造方法来实现，每个参数代表着一个依赖。

**2 Spring中的Bean**

Spring可以被看作一个大型工厂，这个工厂的作用就是生产和管理Spring容器中的Bean。

[2.1 Bean的配置](E:/Web%E7%BC%96%E7%A8%8B%E6%8A%80%E6%9C%AF2018%E5%B9%B4/%E6%A8%A1%E5%9D%974/Spring%E6%A1%86%E6%9E%B6%E6%8A%80%E6%9C%AF/b1) [2.2 Bean的实例化](E:/Web%E7%BC%96%E7%A8%8B%E6%8A%80%E6%9C%AF2018%E5%B9%B4/%E6%A8%A1%E5%9D%974/Spring%E6%A1%86%E6%9E%B6%E6%8A%80%E6%9C%AF/b2) [2.3 Bean的作用域](E:/Web%E7%BC%96%E7%A8%8B%E6%8A%80%E6%9C%AF2018%E5%B9%B4/%E6%A8%A1%E5%9D%974/Spring%E6%A1%86%E6%9E%B6%E6%8A%80%E6%9C%AF/b3) [2.4 Bean的装配方式](E:/Web%E7%BC%96%E7%A8%8B%E6%8A%80%E6%9C%AF2018%E5%B9%B4/%E6%A8%A1%E5%9D%974/Spring%E6%A1%86%E6%9E%B6%E6%8A%80%E6%9C%AF/b3)

**2.1 Bean的配置**

  在Spring中，XML配置文件的根元素是<beans>，<beans>中包含多个<bean>子元素，每一个<bean>子元素定义了一个Bean，并描述了该Bean和被装配到Spring容器中。

  在配置文件中，通常一个普通的Bean只需要定义id（或name）和class两个属性即可。

**2.2 Bean的实例化**

在Spring中，要想使用容器中的Bean，需要实例化。实例化Bean有三种方式：  
A.构造器实例化（最常用）

B.静态工厂实例化（有兴趣自学）C.实例工厂方式实例化（有兴趣自学）

**A.构造器实例化**

构造器实例化指Spring容器通过Bean对应类中默认的无参数方法来实例化Bean。

例：构造器实例化

**(1)** 在spring01的Web项目下路径src/main/java下 创建一个**jmu.constructor**包， 在该包中创建Bean1类。

|  |
| --- |
| package mu.constructor;  public class Bean1 { } |

**(2)** 在applicationContext.xml中， 在配置文件中定义一个id为bean1的Bean， 并通过class属性指定其对应的实现类为Bean1。

|  |
| --- |
| **<bean id="bean1" class="jmu.constructor.Bean1" />**  </beans> |

**(3)** 创建测试类**Bean1Test**， 来测试构造器能否实例化Bean。

|  |
| --- |
| public class Bean1Test {   **@Test**  public void testBean1(){  ApplicationContext app =  new ClassPathXmlApplicationContext("applicationContext.xml");  Bean1 bean1 = (Bean1) app.getBean("bean1");  System.out.println(bean1.toString());  } } |

**2.3 Bean的作用域**

**A.Singleton作用域**

  Bean的作用域为singleton时，Spring容器就只会存在一个共享的Bean实例，并且所有对Bean的请求，只要id与该Bean的id属性相匹配，就会返回同一个Bean实例。是Spring容器的默认作用域。

例：singleton作用域

**(1)** 在spring01的Web项目下路径src/main/java下 创建一个**jmu.scope**包， 在该包中创建**Scope**类。

|  |
| --- |
| package jmu.scope;  public class Scope { } |

**(2)** 在**applicationContext.xml**中， 在配置文件中定义一个id为scope1的Bean， 并通过class属性指定其对应的实现类为Scope。

|  |
| --- |
| **<bean id="scope1" class="jmu.scope.Scope" scope="singleton" />**  </beans> |

**(3)** 创建测试类**ScopeTest**， 来测试singleton的作用域。

|  |
| --- |
| public class **ScopeTest** {   **@Test**  public void **scopeTest1**(){  ApplicationContext app =  new ClassPathXmlApplicationContext("applicationContext.xml");  Scope **scopea** = (Scope) app.getBean("scope1");  Scope **scopeb** = (Scope) app.getBean("scope1");  System.out.println(**scopea**.toString());  System.out.println(**scopeb**.toString());  } |

**B.prototype作用域**

  对需要保持会话状态的Bean应该使用prototype作用域，Spring容器会为每一个对该Bean的请求都创建一个新的实例。

例：prototype作用域

**(1)** 在applicationContext.xml中， 在配置文件中再定义一个id为scope2的Bean， 并通过class属性指定其对应的实现类为Scope。

|  |
| --- |
| **<bean id="scope2" class="jmu.scope.Scope" scope="prototype"/>**  </beans> |

**(2)** 在测试类ScopeTest 中添加测试方法**scopeTest2()**来测试prototype的作用域。

|  |
| --- |
| public class **ScopeTest** {  **@Test**  public void **scopeTest2()**{  ApplicationContext app =  new ClassPathXmlApplicationContext("applicationContext.xml");  Scope scopea = (Scope) app.getBean("scope2");  Scope scopeb = (Scope) app.getBean("scope2");  System.out.println(scopea.toString());  System.out.println(scopeb.toString());  } } |

**2.4 Bean的装配方式** Spring提供了多种形式的Bean装配方式：

A.基于XML的装配 B.基于注解(Annotation)的装配（最常用）C.自动装配 （很少用）

**A 基于XML的装配**

在Spring实例化Bean的过程中，Spring首先会调用Bean的默认构造方法来实例化Bean对象，然后通过反射的方式调用setter方法来注入属性值。因此，设值注入要求一个Bean必须满足以下两点要求：

①Bean类必须提供一个默认的无参构造方法；  
②Bean类必须为需要注入的属性提供对应的setter方法。

例：基于XML的装配方式

**(1)** 在spring01的Web项目下路径src/main/java下 创建一个jmu.assemble包， 在该包中创建Student类。

|  |
| --- |
| package jmu.assemble; import java.util.List;  public class Student {  private String studentId; private String studentName;  private List<String> list;  /\*1.使用构造注入，提供带所有参数的构造方法\*/  public Student(String studentId, String studentName, List<String> list) {  super(); this.studentId = studentId; this.studentName = studentName;  this.list = list; }  /\*2.使用设值注入：提供 默认空参构造方法；为所有属性提供setter方法\*/  public Student(){ super(); }  **SETTER方法 、 toString()方法**  } |

**(2)** 在applicationContext.xml中， 在配置文件中定义一个id为stu1和stu2 的两个Bean，通过XML来装配Bean。

|  |  |
| --- | --- |
| <!-- 构造注入方式装配Student实例 -->  <bean id="stu1" class="cn.jmu.assemble.Student">  <constructor-arg index="0" value="1001" />  <constructor-arg index="1" value="tom" />  <constructor-arg index="2">  <list> <value>"constructorvalue1"</value>  <value>"constructorvalue1"</value>  </list>  </constructor-arg>  </bean> | <!-- 设置注入方式装配Student实例 -->  <bean id="stu2" class="cn.jmu.assemble.Student">  <property name="studentId" value="1002"></property>  <property name="studentName" value="Jane"></property>  <property name="list">  <list>  <value>"listvalue1"</value> <value>"listvalue2"</value>  </list>  </property>  </bean> |

**(3)** 创建测试类**StudentTest**， 来测试XML装配方法。

|  |
| --- |
| public class StudentTest {   **@Test**  public void stuTest1(){  ApplicationContext app =  new ClassPathXmlApplicationContext("applicationContext.xml");  Student stu1 = (Student) app.getBean("stu1");  Student stu2 = (Student) app.getBean("stu2");  System.out.println(stu1.toString());  System.out.println(stu1.toString());  } } |

**B.基于Annotation的装配 （最常用）**

  传统Spring使用XML文件来对bean进行注入，存在两个缺点：  
(1)如果应用中有很多Bean时，会导致XML配置文件过于臃肿，给后续的维护和升级工作带来一定的困难。  
(2)开发中不断在.java和.xml中切换，思维上的不连贯会降低开发的效率  
为此，Spring提供了对Annotation（注解）技术的全面支持。

@Component:仅表示一个组件(Bean)，可以作用在任何层次;

@Repository☆:用于数据访问层(DAO)的类标识为Spring中的Bean;

@Service☆:用于业务层(Service)的类标识为Spring中的Bean;

@Controller☆:用于控制层(Controller)的类标识为Spring中的Bean。

注意：Spring4.0以上版本使用注解方式，需要先向项目导入Spring AOP的JAR包spring-aop-5.\*.\*.RELEASE.jar，否则运行时会报出“java.lang.NoClassDefFound Error:org/springframework/aop/TargetSource”的错误。

例：基于Annotation的装配方式

1. 在spring01的Web项目下路径src/main/java下 创建一个jmu.annotation包， 在该包中创建BookDAO接口，并在接口中定义一个save()方法。

在jmu.annotation包中创建接口实现类 BookDAOImpl， 加上注解@Repository("bookDAO")。

|  |
| --- |
| package jmu.annotation; public interface BookDAO { public void save();} |
| package jmu.annotation; import org.springframework.stereotype.Repository; **@Repository("bookDAO")** public class BookDAOImpl implements BookDAO{  public void save() {System.out.println("bookDAO中的save()方法..."); } } |

1. 在jmu.annotation中， 创建接口BookService,在接口中同样定义一个save()方法。

在jmu.annotation中， 创建接口BookService的实现类 BooktServiceImpl。

|  |
| --- |
| package jmu.annotation; public interface BookService { public void save();} |
| package jmu.annotation; import org.springframework.stereotype.Service; import javax.annotation.Resource; **@Service("bookService")** public class BookServiceImpl implements BookService {  **@Resource(name="bookDAO")**  private BookDAO bookDAO;   public void save() {  this.bookDAO.save();  System.out.println("bookDAO中的save()方法...");  } } |

**(3)** 创建控制器类BookController。

|  |
| --- |
| package jmu.annotation; import org.springframework.stereotype.Controller; import javax.annotation.Resource; **@Controller("bookController")** public class BookController {  **@Resource(name="bookService")**  private BookService bookService;  public void save(){  this.bookService.save();  System.out.println("bookController中的save()方法...");  } } |

**(4)** 在src/main/resources下新创建 一个配置文件**applicationContext2.xml**。 可以从模板applicationContext-aop.xml中复制粘贴。

 在applicationContext2.xml中加上注解扫描包，通知Spring扫描指定包下所有的Bean，进行注解解析

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:context="http://www.springframework.org/schema/context"  xsi:schemaLocation="http://www.springframework.org/schema/beans  http://www.springframework.org/schema/beans/spring-beans.xsd  http://www.springframework.org/schema/context  http://www.springframework.org/schema/context/spring-context-4.3.xsd">  <!-- 使用context命名空间,通知Spring扫描指定包下所有Bean类，进行注解解析 -->  **<context:component-scan base-package="jmu.annotation"/>**  </beans> |

**(5)** 创建测试类BookControllerTest， 来测试注解方法。

|  |
| --- |
| public class **BookControllerTest** {   **@Test**  public void bookTest(){  ApplicationContext app =  new ClassPathXmlApplicationContext("applicationContext2.xml");  BookController bookController =(BookController) app.getBean("**bookController**");  bookController.save();  } } |

**C.自动装配（实际项目中很少用，不讲）**

**缺点：**(1)只能一次装配所有属性，若只希望装配个别属性时，不行。

(2)autowire要么根据类型自动装配，要么根据名称自动装配，不能两者兼而有之。

(3)一般情况下，在实际的项目中很少使用自动装配功能，因为和自动装配功能所带来的好处相比，明确清晰地配置更有说服力。

**3 Spring的AOP技术**

**引例：**加减乘除计算器的日志记录

**(1)** 在spring01的Web项目下路径src/main/java下 创建一个**jmu.cal**包， 在该包中创建**CalDAO**接口，并在接口中定义加减乘除方法。

|  |
| --- |
| package jmu.cal;  public interface CalDAO { int add(int i, int j); int sub(int i, int j);  int mul(int i, int j); int div(int i, int j); } |

**(2)** 在jmu.cal包中 创建CalDAO接口的实现类**CalDAOImpl**实现加减乘除方法。

|  |
| --- |
| package jmu.cal;  import org.springframework.stereotype.Component; **@Component("calDAO")**  public class CalDAOImpl implements CalDAO {  public int add(int i, int j) {  System.out.println("The method **add** begins with["+i+","+j+"]");  int result=i+j;  System.out.println("The method add ends with "+result);  return result; }  } |
| public int sub(int i, int j) {  System.out.println("The method **sub** begins with["+i+","+j+"]");  int result=i-j; System.out.println("The method sub ends with "+result);  return result; } |
| public int mul(int i, int j) {  System.out.println("The method **mul** begins with["+i+","+j+"]");  int result=i\*j; System.out.println("The method mul ends with "+result);  return result; } |
| public int div(int i, int j){  System.out.println("The method **div** begins with["+i+","+j+"]");  int result=i/j; System.out.println("The method div ends with "+result);  return result; } |

1. 在配置文件 applicationContext2.xml中加入 jmu.cal的扫描包。

|  |
| --- |
| <context:component-scan base-package="jmu.cal"/> |

**(4)** 创建测试类CalDAOImplTest。

|  |
| --- |
| public class CalDAOImplTest {  **@Test**  public void calDAOTest(){  ApplicationContext app =  new ClassPathXmlApplicationContext("applicationContext2.xml");  CalDAO cal = (**CalDAO**) app.getBean("**calDAO**");  int result1 = cal.add(3,4);  System.out.println("add-->"+result1);  int result2 = cal.div(10,2);  System.out.println("div-->"+result1);  } } |

**存在的问题：**

☆代码混乱：越来越多的非业务需求(日志和验证等)加入后, 原有的业务方法急剧膨胀. 每个方法在处理核心逻辑的同时还必须兼顾其他多个关注点。  
☆代码分散: 以日志需求为例, 只是为了满足这个单一需求, 就不得不在多个模块（方法）里多次重复相同的日志代码. 如果日志需求发生变化, 必须修改所有模块。

**AOP** 相关概念：

☆切面(Aspect): 横切关注点(跨越应用程序多个模块的功能)被模块化的特殊对象。  
☆通知(Advice): 切面必须要完成的工作。  
☆目标(Target): 被通知的对象。  
☆代理(Proxy): 向目标对象应用通知之后创建的对象。  
☆连接点（Joinpoint）：程序执行的某个特定位置——物理意义上的概念。  
☆切点（pointcut）：每个类都拥有多个连接点--逻辑意义上的概念。

主流AOP技术——基于注解的声明式AspectJ

|  |  |
| --- | --- |
| @Aspect | 定义一个切面 |
| @Pointcut | 定义切入点的表达式。还需要定义一个返回值为void，方法体为空的方法 |
| @Before | 用于定义前置通知，通常需要指定一个value属性值（切入点） |
| @After | 用于定义最终final通知，不管是否异常，该通知都会执行，通常需要指定一个value属性值（切入点） |
| @AfterReturning | 用于定义后置通知，通常需要指定一个value属性值（切入点） ，一个returning属性值（方法返回值） |
| @Around | 用于定义环绕通知，通常需要指定一个value属性值（切入点） |
| @AfterThrowing | 用于定义异常通知，通常需要指定一个value属性值（切入点） ，一个throwing属性值（方法抛出的异常） |

**如果是基于XML的声明式AspectJ的配置文件applicationContext.xml**

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"...**aop\context** t-4.3.xsd">  <!-- 1 目标类 -->  <bean id="cal" class="cn.cal.aop.CalDAOImpl" />  <!-- 2 切面类 -->  <bean id="loggingAspect" class="cn.cal.aop.LoggingAspect" />  <aop:config>  <!--3 配置切面 -->  <aop:aspect id="aspect1" ref="loggingAspect">  <!--4 配置切入点 -->  <aop:pointcut expression="execution(\* cn.cal.aop.CalDAOImpl.\*(..))" id="myPointCut" />  <!-- 5 配置通知 -->  <aop:before method="beforeMethod" pointcut-ref="myPointCut" />  <aop:after method="afterMethod" pointcut-ref="myPointCut" />  <aop:after-returning method="afterReturning" pointcut-ref="myPointCut" returning="result" />  <aop:after-throwing method="afterThrowing" pointcut-ref="myPointCut" throwing="e" />  <aop:around method="aroundMethod" pointcut-ref="myPointCut" />  </aop:aspect>  </aop:config>  </beans> |

**例：**将验证参数、前置日志、后置日志“织入”四则运算中。——基于注解的声明式AspectJ

**(1)** 在spring01的Web项目下路径src/main/java下 创建一个**jmu.calaop**包， 在该包中同样创建CalDAO2接口，并在接口中定义加减乘除方法。

|  |
| --- |
| package jmu.calaop;  public interface **CalDAO2** { int add(int i, int j); int sub(int i, int j); int mul(int i, int j); int div(int i, int j);} |

**(2)** 在jmu.calaop包中 创建CalDAO2接口的实现类**CalDAO2Impl** 实现加减乘除的核心方法。

|  |
| --- |
| package jmu.calaop;  import org.springframework.stereotype.Component; import org.springframework.stereotype.Repository;  @Repository("**cal2**") **等价于**<bean id="cal" class="cn.cal.aop.CalDAOImpl" />  public class **CalDAO2Impl** implements CalDAO2 {  public int add(int i, int j) {int result=i+j; return result;}  public int sub(int i, int j) {int result=i-j; return result;}  public int mul(int i, int j) {int result=i\*j; return result;}  public int div(int i, int j) {int result=i/j; return result; }} |

**(3)** 在jmu.calaop包中 创建日志切面类LoggingAspect。

  A.切面首先是一个IoC中的Bean，即加入@Component;

  B.加入切面注解@Aspect；

  C.在类中声明各种通知（声明一个方法，在方法前加@Before或@After或@AfterReturning或@Around或@AfterThrowing注解）；

  D.可以在通知方法中声明一个类型为JoinPoint（org.aspectj.lang.JoinPoint;而非org.aopalliance.intercept）参数，即可访问连接点的细节，如方法名和参数值

|  |
| --- |
| package jmu.calaop; import java.util.Arrays;  import org.aspectj.lang.JoinPoint;  import org.aspectj.lang.annotation.After\AfterReturning\AfterThrowing  \Aspect\Before\Pointcut;  import org.springframework.core.annotation.Order;  import org.springframework.stereotype.Component;  //可以使用 @Order 注解指定切面的优先级, 值越小优先级越高  **@Order(2)**  **@Aspect 等价于** <aop:aspect id="aspect1" ref="loggingAspect">  **@Component**  public class LoggingAspect {  //定义切入点表达式 等价于  //<aop:pointcut expression="execution(\* cn.cal.aop.CalDAOImpl.\*(..))" id="myPointCut" />  **@Pointcut("execution(\* jmu.calaop.CalDAO2Impl.\*(..))")**  //使用一个返回值为void、方法体为空的方法来命名切入点  private void myPointCut(){}  } |
| //在cn.cal.aop.CalDAOImpl的每一个方法开始之前执行一段代码  //等价于<aop:before method="beforeMethod" pointcut-ref="myPointCut" />  **@Before("myPointCut()")**  public void beforeMethod(JoinPoint joinPoint){  String methodName=joinPoint.getSignature().getName();  Object[] args=joinPoint.getArgs();  System.out.println("The method "+ methodName+" begin with "+Arrays.asList(args));  } |
| //在方法执行之后执行的代码. 无论该方法是否出现异常  //等价于<aop:after method="afterMethod" pointcut-ref="myPointCut" />  **@After("myPointCut()")**  public void afterMethod(JoinPoint joinPoint){  String methodName = joinPoint.getSignature().getName();  System.out.println("The method " + methodName + " ends");  } |
| //在方法法正常结束受执行的代码,返回通知是可以访问到方法的返回值的!  //等价于<aop:after-returning method="afterReturning" pointcut-ref="myPointCut" returning="result" />  **@AfterReturning(value="myPointCut()",returning="result")**  public void afterReturning(JoinPoint joinPoint, Object result){  String methodName = joinPoint.getSignature().getName();  System.out.println("The method " + methodName + " ends with " + result);  } |
| //在目标方法出现异常时会执行的代码  //等价于<aop:after-throwing method="afterThrowing" pointcut-ref="myPointCut" throwing="e" />  **@AfterThrowing(value="myPointCut()",throwing="e")**  public void afterThrowing(JoinPoint joinPoint, Exception e){  String methodName = joinPoint.getSignature().getName();  System.out.println("The method " + methodName + " occurs excetion:" + e);  } |
| /\* \* 环绕通知需要携带 ProceedingJoinPoint 类型的参数.  \* 环绕通知类似于动态代理的全过程: ProceedingJoinPoint 类型的参数可以决定是否执行目标方法.  \* 且环绕通知必须有返回值, 返回值即为目标方法的返回值 \*/  //等价于<aop:around method="aroundMethod" pointcut-ref="myPointCut" />  **@Around("myPointCut()")**  public Object aroundMethod(ProceedingJoinPoint pjd){  Object result = null;  String methodName = pjd.getSignature().getName();  try {//前置通知  System.out.println("The method " + methodName + " begins with " + Arrays.asList(pjd.getArgs()));  result = pjd.proceed();//执行目标方法  //返回通知  System.out.println("The method " + methodName + " ends with " + result);  } catch (Throwable e) {//异常通知    System.out.println("The method " + methodName + " occurs exception:" + e);  throw new RuntimeException(e); }  //后置通知  System.out.println("The method " + methodName + " ends");  return result; } |

**(4)**在jmu.calaop包下创建类ValidateAspect，把横切关注点(验证参数)的代码抽象到该类中。

|  |
| --- |
| Package jmu.calaop; import java.util.Arrays; import org.aspectj.lang.JoinPoint;  import org.aspectj.lang.annotation.Aspect\Before\Pointcut;  import org.springframework.core.annotation.Order;  import org.springframework.stereotype.Component;  **@Order(1)**  **@Aspect**  **@Component**  public class ValidateAspect {  //定义切入点表达式  **@Pointcut("execution(\* jmu.calaop.CalDAO2Impl.\*(..))")**  //使用一个返回值为void、方法体为空的方法来命名切入点  private void myPointCut(){}  **@Before("myPointCut()")**  public void validateArgs(JoinPoint joinPoint){  System.out.println("-->validate:" + Arrays.asList(joinPoint.getArgs()));}  } |

**（5）applicationContext2.xml**

A.在配置文件中加入AOP命名空间;

B.扫描包。<context:component-scan base-package="jmu.calaop"/>

C.基于注解方式的配置。<aop:aspectj-autoproxy/>

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="beans" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:aop xmlns:context xsi:schemaLocation="/beans aop context">  <!-- 使用context命名空间,通知Spring扫描指定包下所有Bean类，进行注解解析 -->  **<context:component-scan base-package="jmu.calaop"/>**  <!-- 配置自动为匹配 aspectJ 注解的 Java 类生成代理对象 -->  **<aop:aspectj-autoproxy></aop:aspectj-autoproxy>**  </beans> |

**(6)** 创建测试类CalDAO2ImplTest。

|  |
| --- |
| public class CalDAO2ImplTest {  **@Test**  public void cal2Test(){  ApplicationContext app =  new ClassPathXmlApplicationContext("applicationContext2.xml");  CalDAO2 cal2 = (**CalDAO2**)app.getBean("**cal2**");  cal2.add(3,4);  cal2.div(10,2);  } } |